CS 5830 Homework 5

`Instructor: Thomas Ristenpart`
`TAs: Andres Fabrega, Sanketh Menda, and Dhrumil Patel`
`Due: December 5, 2022`

Submission and Grading

Submit your written answers to Gradescope and your code answers to CMSX. You must submit appropriate answers to the both submission sites for full credit.

Problems in this assignment are of two types:

1. *Attacking cryptography.* Given an incomplete template file, you need to fill in the missing pieces to mount an attack, e.g. retrieving a flag or forging a ciphertext.
2. *Talking cryptography.* Talk us through how you attacked a given cryptographic primitive or how you would hypothetically attack it.

The exact materials and the format you would need to submit them in for a given problem are indicated at the end of the problem.

Late Days

Homeworks are due on the due date by 11:59pm EST. You can use in total 3 late days throughout the semester.

Academic Integrity

Feel free to refer to external materials, but as usual please acknowledge them in your writeup in accordance with the Cornell Code of Academic Integrity.

Challenge Server Availability

To query the server, you'll use the same username and API key throughout this semester. Your username is your Cornell NetID (like `sm2289`) and you can find your assigned API key on CMSX, in the *Grading Comments & Requests* section of the placeholder assignment called `API_keys`.

We made a best effort to ensure the reliability of the challenge servers, but unfortunately, due to the nature of software and the internet, there may be disruptions. If you run into a disruption (the server is not responding, it is providing an error code, etc.), please do not panic, and reach out to the TAs (`@sanketh`, `@Andres`, or `@Dhrumil Patel`) on Slack. If this happens within 24 hours of the deadline, we will extend the deadline to accommodate for the inconvenience.

Verbosity for Written Answers

Generally, we prefer concise answers.

For a question like "Explain how you mount the attack?" we are expecting a high-level, but precise description of the attack in 3-10 sentences. Here is an example of a good answer.

> I started with an initial guess of the 16-byte all-zeros message. Then for each of the 16 bytes, I did X. Then did Y. Since the success of this is not guranteed, I repeated the process 20 times and aggregated. With the confidence, I did Z to recover the `FLAG`.

For a question like "Estimate the number of queries your attack issues?" we want you to think about how many queries you *really* need, and expect a high-level answer with ballpark numbers. Here is an example of a good answer.

> My attack does 5 queries initially for [purpose]. Then it does 1 queries per `FLAG` byte. In sum, for a flag of length N, it needs 1 + 5/N queries per `FLAG` byte.

Flag Notes

Each problem includes a description of the expected flag. For instance, it may say that it has 10 alphanumeric characters. Feel free to use that to restrict your search space, and to verify that you found the right flag (if you find something that meets the flag description, it is overwhelmingly likely that you found the right flag).

During grading, we will run your code against a different flag that nevertheless meets the description. So please make sure that you are not hardcoding assumptions regarding your specific flag.

Submission Server Availability

We use gradescope and CMSX for accepting submissions. There is a chance that these services might be down during the submission period. If you run into this, please send your submission, with a screenshot of the disruption, to the instructor, via email, at `ristenpart@cornell.edu`.

Question 1. Invisible Salamanders in GCM

Sam recently quit Twitter, and wanted to find community on Mastodon. Rather than join an existing instance like infosec.exchange or octodon.social, they wanted to run their own instance. Their initial plan was to just run the official implementation. But, after they found out its direct messages (DMs) were just "private posts", they decided to implement their own end-to-end encrypted (E2EE) DMs.

For the underlying protocol, they chose Signal's libsignal, which takes them most of the way. But, they wanted to add in abuse reporting; i.e., users should be able to report abusive messages (like spam, hate speech, and harassment) so that moderators can take action. This is non-trivial in the E2EE setting because the platform doesn't know which messages were sent over the platform.

They solve this problem using "message franking" as introduced in Facebook's Secret Conversations whitepaper. The high-level idea is that the platform HMACs the ciphertext under a platform key, and then gives that HMAC tag to the recipient. Now, if the recipient wants to report the message as abusive, they provide the ciphertext with the HMAC tag as proof that it was sent over the platform.

So, Sam launches their mastodon instance with E2EE direct messaging, and it is a huge hit, it makes it to the top of hacker news, and it is being hugged to death by all the usage. Upon inspection, they notice that >90% of the traffic going through the messaging service is attachments like images, audio, and video. So they decide to spin up a distributed CDN to reduce this load. It works as follows: the sender encrypts their attachment m using a random key k and a random nonce n, to get a ciphertext C. Then, uploads the ciphertext C to the CDN, which for a short time hosts the content which can be retrieved with its hash SHA256(C). Then, the sender sends the context (k, n) and a hash of the ciphertext SHA256(C) to the recipient, who can fetch the attachment ciphertext C from the CDN and decrypt it using the provided context.

At the same time, they notice that their moderation console is clogged with duplicate reports, so they implement deduplication. For attachments, for every unique hash SHA256(C) reported, the system attaches the corresponding plaintext to the report.

Notice that this is a subtle bug with this deduplication approach, the message m is a function of the ciphertext C, and the decryption context (k, n), but the deduplication is done only on C. So, if an attacker can produce a ciphertext C that decrypts under different decryption contexts (k_1, n_1) and (k_2, n_2), then they may be able to spoof the moderator's input. The question asks to produce such a ciphertext for AES128-GCM.

Problem Oracle. We are given two oracles,

1. `getKeyNonceMessage` takes nothing and returns a key, nonce, and message.
2. `verifyCollision` takes a ciphertext and two decryption contexts and returns if they are valid.

The Goal. The goal is to construct a valid collision for AES128-GCM. Skim Sophie Schmieg's blog post for an overview of the attack. Next, skim the GCM specification to understand the specifics of GCM. Now, re-read the blog post and make notes on how to do the attack. You may also find Section 3.1 of Dodis et al. (2019) helpful.

The provided template `hw5_q1_gcm_salamanders.py` has a partial implementation of the attack, and the provided module `gf128.py` has helpful subroutines.

Query Estimates. This attack is expected to take up to a few minutes of CPU time.

What to Submit.

- (5 pts) *Source code.* Copy the files `hw5_q1_gcm_salamanders.py`, `gf128.py`, and `q1_oracles.py` into your homeworks' folder and complete the TODOs to conduct the attack. Submit `hw5_q1_gcm_salamanders.py` to CMSX.
- (5 pts) *Written answer.* Submit answers to the following questions to Gradescope:
    1. (3 pts) Explain how you mount the attack?
    2. (2 pts) Estimate the number of queries your attack issues?
- (1 pts extra credit) *Written answer.* Submit answers to the following questions to Gradescope:
    1. (1 pts) Describe another real-world instance where this attack can be devastating.

Question 2. Compression-Oracle Attacks

Introduction. Justin and their friends have recently become very security conscious, so they decide to encrypt every message sent between them using AES-CTR. After experimenting with this system for a while, Justin finds themselves overwhelmed by the size of the ciphertexts! So, they decide to compress messages before encrypting them using zlib, a popular library for this purpose. In addition, Justin and their friends agree on a `flag` that they prepend to every message, naively hoping that it serves as a "tag" to make sure that messages have not been tampered with[1]. Putting it all together, the scheme looks like this:

`AES128-CTR-Encrypt(K, ZLIB(FLAG||m))`

where `FLAG` is of the form `ZLIB{[5 random bytes in hex]}`, like `ZLIB{ffffffffff}`.

You do not need to understand the low-level details of how zlib works for this problem, but the key idea is that it replaces repeated sequences with pointers to prior occurrences of them. For example, in `abcde123abcdegh`, the second copy of `abcde` gets replaced by `(9, 5, g)`, denoting that the next 5 characters are a copy of the prefix located 9 characters before it, and that the next character to compress is `g`. Skim this article for a good description of the algorithm, and pay particular attention to the example in section 2.

Eve hears about their new scheme, and remembers from her security class that compressing before encrypting tends to be a bad idea! In particular, she remembers the CRIME attack on TLS, which exploits compression to learn cookies one byte at a time. If Eve has access to an oracle of this scheme, she can use a similar attack to find the flag! Let's explore this further.

Eve knows that `FLAG` starts with `ZLIB{`. So, if she injects a message containing just this string, it will get fully compressed with the first copy of it in `FLAG`, and replaced by a "pointer", as described above. What happens if she inserts an extra byte, like `ZLIB{X`? If `X` matches the first byte of the flag, her entire message will get completely compressed again; conversely, if her guess is incorrect, `X` will not be included in the aforementioned pointer, meaning that the ciphertext is now longer than in the first case! So, if she tests all possible values for this byte, the correct guess will result in a slightly shorter ciphertext. She can proceed in this fashion for every byte, and learn the entire `FLAG`. Note that it could be the case that multiple byte values result in the same minimal ciphertext size (try to think why!), in which case you need to test all of these in the next iteration.

Query Estimates. This attack should ideally take a few thousand queries, and a few minutes of CPU time.

Flag Format. `ZLIB{[5 random bytes in hex]}`, like `ZLIB{ffffffffff}`.

What to Submit.

- (2 pts) *Source code.* Copy the file `hw5_q2_compression_oracle.py` into your homeworks' folder and complete the TODOs to conduct the attack. Submit `hw5_q2_compression_oracle.py` to CMSX.
- (3 pts) *Captured flag.* Put the captured `FLAG`, just the `FLAG` no quotes or special characters in text (not hex) in a new text file `hw5_compression_oracle.txt` and submit it to CMSX.
- (5 pts) *Written answer.* Submit your answers to the following questions to Gradescrope:

[1]Justin's idea does not really serve as a proper MAC, but this is outside the scope of this problem.

1. (3 pts) Explain how you found the flag?
2. (2 pts) Estimate the number of queries your attack issues per byte of `FLAG`?

- (1 pts extra credit) *Written answer.* Submit answers to the following questions to Gradescope:
    1. (1 pts) Is it possible to design a compression algorithm that would not be susceptible to such an attack? If so, describe how. If not, explain why not.

Evaluation

We would love it if you could provide us feedback to improve future homeworks. Please provide answers to the following questions on Gradescrope.

- Did you find the homework easy, appropriately difficult, or too difficult?
- How many hours did you spent on this homework?
- Did you feel that the coding was too much, appropriate, or not enough?

And feel free to include additional feedback (on the homework or logistics in general). Also, this is not a tight deadline, we'd be delighted to hear your feedback via Slack before or after the deadline.